APPLICATION NOTE

AP-415

July 1988

83C51FA/FB PCA Cookbook

BETSY JONES
ECO APPLICATIONS ENGINEER

Order Number: 270609-001

This application note illustrates the different functions of the Programmable Counter Array (PCA) which are available on the 83C51FA and 83C51FB. Included are cookbook samples of code in typical applications to simplify the use of the PCA. Since all the examples are written in assembly language, it is assumed the reader is familiar with ASM51. For further information on these products or ASM51 refer to the Embedded Controller Handbook (Vol. 1).

PCA OVERVIEW

The major new feature on the 83C51FA and 83C51FB is the Programmable Counter Array. The PCA provides more timing capabilities with less CPU intervention than the standard timer/counters. Its advantages include reduced software overhead and improved accuracy.

The PCA consists of a dedicated timer/counter which serves as the time base for an array of five compare/capture modules. Figure 1 shows a block diagram of the PCA. Notice that the PCA timer and modules are all 16-bits. If an external event is associated with a module, that function is shared with the corresponding Port 1 pin. If the module is not using the port pin, the pin can still be used for standard I/O.

Each of the five modules can be programmed in any one of the following modes:

- Rising and/or Falling Edge Capture
- Software Timer
- High Speed Output
- Watchdog Timer (Module 4 only)
- Pulse Width Modulator.

All of these modes will be discussed later in detail. However, let's first look at how to set up the PCA timer and modules.

PCA TIMER/COUNTER

The timer/counter for the PCA is a free-running 16-bit timer consisting of registers CH and CL (the high and low bytes of the count values). It is the only timer which can service the PCA. The clock input can be selected from the following four modes:

- oscillator frequency ÷ 12 (Mode 0)
- oscillator frequency ÷ 4 (Mode 1)
- Timer 0 overflows (Mode 2)
- external input on P1.2 (Mode 3)

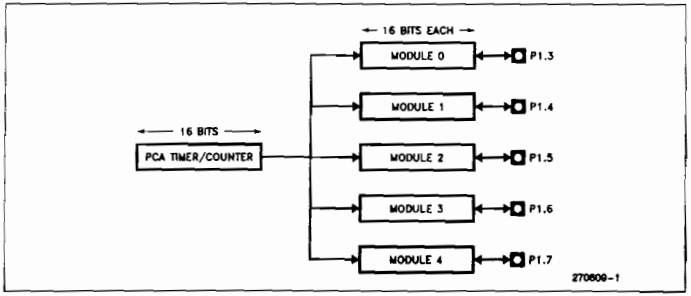


Figure 1. PCA Timer/Counter and Compare/Capture Modules

The table below summarizes the various clock inputs for each mode at two common frequencies. In Mode 0, the clock input is simply a machine cycle count, whereas in Mode 1 the input is clocked three times faster. In Mode 2, Timer 0 overflows are counted allowing for a range of slower inputs to the timer. And finally, if the input is external the PCA timer counts 1-to-0 transitions with the maximum clock frequency equal to 1/8 x oscillator frequency.

Table 1. PCA Timer/Counter Inputs

PCA Timer/Counter Mode	Clock Increments	
	12 MHz	16 MHz
Mode 0: fosc / 12	1 μsec	0.75 μsec
Mode 1: fosc / 4	330 nsec	250 nsec
Mode 2*: Timer 0 Overflows		
Timer 0 programmed in:		
8-bit mode	256 μsec	192 μsec
16-bit mdoe	65 msec	49 msec
8-bit auto-reload	1 to 255 μsec	0.75 to 191 μsec
Mode 3: External Input MAX	0.66 μsec	0.50 μsec

*In Mode 2, the overflow interrupt for Timer 0 does not need to be enabled.

Special Function Register CMOD contains the Count Pulse Select bits (CPS1 and CPS0) to specify the PCA timer input. This register also contains the ECF bit which enables an interrupt when the counter overflows. In addition, the user has the option of turning off the PCA timer during Idle Mode by setting the Counter Idle bit (CIDL). This can further reduce power consumption by an additional 30%.

CMOD: Counter Mode Register

CIDL	WDTE	—	—	—	CPS1	CPS0	ECF

Address = 0D9H
Not Bit Addressable

Reset Value = 00XX X000B

NOTE:
The user should write 0s to unimplemented bits. These bits may be used in future MCS-51 products to invoke new features, and in that case the inactive value of the new bit will be 0. When read, these bits must be treated as don't-cares.

Table 2 lists the values for CMOD in the four possible timer modes with and without the overflow interrupt enabled. This list assumes that the PCA will be left running during Idle Mode.

Table 2. CMOD Values

PCA Count Pulse Selected	CMOD value	
	without interrupt enabled	with interrupt enabled
Internal clock, Fosc/12	00 H	01 H
Internal clock, Fosc/ 4	02 H	03 H
Timer 0 overflow	04H	05 H
External clock at P1.2	06 H	07 H

The CCON register shown below contains the Counter Run bit (CR) which turns the timer on or off. When the PCA timer overflows, the Counter Overflow bit (CF) gets set. CCON also contains the five event flags for the PCA modules. The purpose of these flags will be discussed in the next section.

CCON: Counter Control Register

CF	CR	—	CCF4	CCF3	CCF2	CCF1	CCF0

Address = 0D8H
Bit Addressable

Reset Value = 00X0 0000B

The PCA timer registers (CH and CL) can be read and written to at any time. However, to read the full 16-bit timer value simultaneously requires using one of the PCA modules in the capture mode and toggling a port pin in software. More information on reading the PCA timer is provided in the section on the Capture Mode.

COMPARE/CAPTURE MODULES

Each of the five compare/capture modules has a mode register called CCAPMn (n = 0,1,2,3,or 4) to select which function it will perform. Note the ECCFn bit which enables an interrupt to occur when a module's event flag is set.

CCAPMn: Compare/Capture Mode Register

—	ECOMn	CAPPn	CAPNn	MATn	TOGn	PWMn	ECCFn

Address = 0DAH (n = 0)
0DBH (n = 1)
0DCH (n = 2)
0DDH (n = 3)
0DEH (n = 4)

Reset Value = X000 0000B

Table 3 lists the CCAPMn values for each different mode with and without the PCA interrupt enabled; that is, the interrupt is optional for all modes. However, some of the PCA modes require software servicing. For example, the Capture modes need an interrupt so that back-to-back events can be recognized. Also, in most applications the purpose of the Software Timer mode is to generate interrupts in software so it would be useless not to have the interrupt enabled. The PWM mode, on the other hand, does not require CPU intervention so the interrupt is normally not enabled.

Table 3. Compare/Capture Mode Values

Module Function	CCAPMn Value	
	without interrupt enabled	with interrupt enabled
Capture Positive only	20H	21 H
Capture Negative only	10H	11 H
Capture Pos. or Neg.	30H	31 H
Software Timer	48H	49 H
High Speed Output	4C H	4D H
Watchdog Timer	48 or 4C H	—
Pulse Width Modulator	42 H	43H

It should be mentioned that a particular module can change modes within the program. For example, a module might be used to sample incoming data. Initially it could be set up to capture a falling edge transition. Then the same module can be reconfigured as a software timer to interrupt the CPU at regular intervals and sample the pin.

Each module also has a pair of 8-bit compare/capture registers (CCAPnH, CCAPnL) associated with it. These registers are used to store the time when a capture event occurred or when a compare event should occur. Remember, event times are based on the free-running PCA timer (CH and CL). For the PWM mode, the high byte register CCAPnH controls the duty cycle of the waveform.

When an event occurs, a flag in CCON is set for the appropriate module. This register is bit addressable so that event flags can be checked individually.

CCON: Counter Control Register

CF	CR	—	CCF4	CCF3	CCF2	CCF1	CCF0

Address = 0D8H — Reset Value = 00X0 0000B
Bit Addressable

These five event flags plus the PCA timer overflow flag share an interrupt vector as shown below. These flags are not cleared when the hardware vectors to the PCA interrupt address (0033H) so that the user can determine which event caused the interrupt. This also allows the user to define the priority of servicing each module.

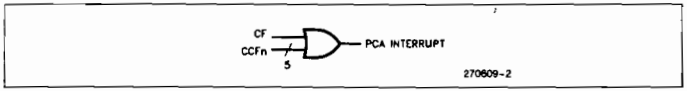


Figure 2. PCA Interrupt

An additional bit was added to the Interrupt Enable (IE) register for the PCA interrupt. Similarly, a high priority bit was added to the Interrupt Priority (IP) register.

IE: Interrupt Enable Register

EA	EC	ET2	ES	ET1	EX1	ET0	EX0

Address = 0A8H — Reset Value = 0000 0000B
Bit Addressable

IP: Interrupt Priority Register

—	PPC	PT2	PS	PT1	PX1	PT0	PX0

Address = 0B8H — Reset Value = X000 0000B
Bit Addressable

Remember, each of the six possible sources for the PCA interrupt must be individually enabled as well—in the CCAPMn register for the modules and in the CCON register for the timer.

CAPTURE MODE

Both positive and negative transitions can trigger a capture with the PCA. This allows the PCA flexibility to measure periods, pulse widths, duty cycles, and phase differences on up to five separate inputs. This section gives examples of all these different applications.

Figure 3 shows how the PCA handles a capture event. Using Module 0 for this example, the signal is input to P1.3. When a transition is detected on that pin, the 16-bit value of the PCA timer (CH,CL) is loaded into the capture registers (CCAP0H,CCAP0L). Module 0's event flag is set and an interrupt is flagged. The interrupt will then be generated if it has been properly enabled.

In the interrupt service routine, the 16-bit capture value must be saved in RAM before the next event capture occurs; a subsequent capture will write over the first capture value. Also, since the hardware does not clear the event flag, it must be cleared in software.

The time it takes to service this interrupt routine determines the resolution of back-to-back events with the same PCA module. To store two 8-bit registers and clear the event flag takes at least 9 machine cycles. That includes the call to the interrupt routine. At 12 MHz, this routine would take less than 10 microseconds. However, depending on the frequency and interrupt latency, the resolution will vary with each application.

Measuring Pulse Widths

To measure the pulse width of a signal, the PCA module must capture both rising and falling edges (see Figure 4). The module can be programmed to capture either edge if it is known which edge will occur first. However, if this is not known, the user can select which edge will trigger the first capture by choosing the proper mode for the module.

Listing 1 shows an example of measuring pulse widths. (It's assumed the incoming signal matches the one in Figure 4.) In the interrupt routine the first set of capture values are stored in RAM. After the second capture, a subtraction routine calculates the pulse width in units of PCA timer ticks. Note that the subtraction does not have to be completed in the interrupt service routine. Also, this example assumes that the two capture events will occur within 2^{16} counts of the PCA timer, i.e. rollovers of the PCA timer are not counted.

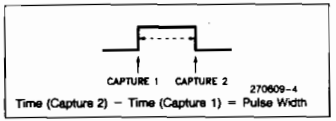


Figure 4. Measuring Pulse Width

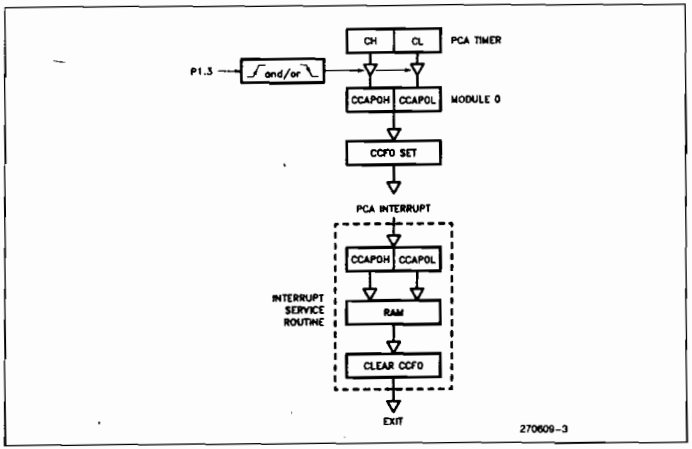


Figure 3. PCA Capture Mode (Module 0)

Listing 1. Measuring Pulse Widths

```
; RAM locations to store capture values
    CAPTURE          DATA    30H
    PULSE_WIDTH      DATA    32H
    FLAG             BIT     20H.0
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:                    ; Initialize PCA timer
    MOV CMOD, #00H           ; Input to timer = 1/12 X Fosc
    MOV CH, #00H
    MOV CL, #00H
;
;   Initialize Module 0 in capture mode
    MOV CCAPM0, #21H         ; Capture positive edge first
                             ; for measuring pulse width
;
    SETB EC                  ; Enable PCA interrupt
    SETB EA
    SETB CR                  ; Turn PCA timer on
    CLR FLAG                 ; clear test flag
;
;   ************************************************************************
;                Main program goes here
;   ************************************************************************
;
;   This example assumes Module 0 is the only PCA module
;   being used. If other modules are used, software must
;   check which module's event caused the interrupt.
;
PCA_INTERRUPT:
    CLR CCF0                 ; Clear Module 0's event flag
    JB FLAG, SECOND_CAPTURE  ; Check if this is the first
                             ; capture or second
FIRST_CAPTURE:
    MOV CAPTURE, CCAP0L      ; Save 16-bit capture value
    MOV CAPTURE+1, CCAP0H    ; in RAM
    MOV CCAPM0, #11H         ; Change module to now capture
                             ; falling edges
    SETB FLAG                ; Signify 1st capture complete
    RETI
;
SECOND_CAPTURE:
    PUSH ACC
    PUSH PSW
    CLR C
    MOV A, CCAP0L            ; 16-bit subtract
    SUBB A, CAPTURE
    MOV PULSE_WIDTH, A       ; 16-bit result stored in
    MOV A, CCAP0H            ; two 8-bit RAM locations
    SUBB A, CAPTURE+1
    MOV PULSE_WIDTH+1, A
    ;
    MOV CCAPM0, #21H         ; Optional--needed if user wants to
    CLR FLAG                 ; measure next pulse width
    POP PSW
    POP ACC
    RETI
```


Measuring Periods

Measuring the period of a signal with the PCA is similar to measuring the pulse width. The only difference will be the trigger source for the capture mode. In Figure 5, rising edges are captured to calculate the period. The code is identical to Listing 1 except that the capture mode should not be changed in the interrupt routine. The result of the subtraction will be the period.

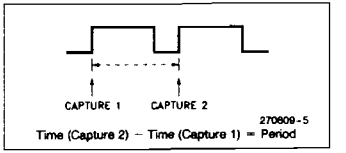


Figure 5. Measuring Period

Measuring Frequencies

Measuring a frequency with the PCA capture mode involves calculating a sample time for a known number of samples. In Figure 6, the time between the first capture and the "Nth" capture equals the sample time T. Listing 2 shows the code for N = 10 samples. It's assumed that the sample time is less than 2^{16} counts of the PCA timer.

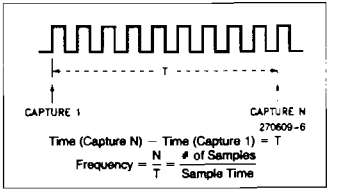


Figure 6. Measuring Frequency

Listing 2. Measuring Frequencies

```
; RAM locations to store capture values
   CAPTURE        DATA      30H
   PERIOD         DATA      32H
   SAMPLE_COUNT   DATA      34H
   FLAG           BIT       20H.0
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:
; Initialization of PCA timer, Module 0, and interrupt is the
; same as in Listing 1. Also need to initialize the sample
; count.
;
   MOV SAMPLE_COUNT, #10D   ; N = 10 for this example
;
;************************************************************
;                  Main program goes here                   ;
;************************************************************
;
;  This code assumes only Module 0 is being used.
PCA_INTERRUPT:
   CLR CCF0                 ; Clear module 0's event flag
   JB FLAG, NEXT_CAPTURE
;
FIRST_CAPTURE:
   MOV CAPTURE, CCAP0L
   MOV CAPTURE+1, CCAP0H
   SETB FLAG                ; Signify first capture complete
   RETI
;
NEXT_CAPTURE:
   DJNZ SAMPLE_COUNT, EXIT
   PUSH ACC
   PUSH PSW
   CLR C
   MOV A, CCAP0L            ; 16-bit subtraction
   SUBB A, CAPTURE
   MOV PERIOD, A
   MOV A, CCAP0H
   SUBB A, CAPTURE+1
   MOV PERIOD+1, A
   ;
   MOV SAMPLE_COUNT, #10D   ; Reload for next period
   CLR FLAG
   POP PSW
   POP ACC
EXIT:
   RETI
```


The user may instead want to measure frequency by counting pulses for a known sample time. In this case, one module is programmed in the capture mode to count edges (either rising or falling), and a second module is programmed as a software timer to mark the sample time. An example of a software timer is given later. For information on resolution in measuring frequencies, refer to Article Reprint AR-517, "Using the 8051 Microcontroller with Resonant Transducers," in the Embedded Controller Handbook.

Measuring Duty Cycles

To measure the duty cycle of an incoming signal, both rising and falling edges need to be captured. Then the duty cycle must be calculated based on three capture values as seen in Figure 7. The same initialization routine is used from the previous example. Only the PCA interrupt service routine is given in Listing 3.

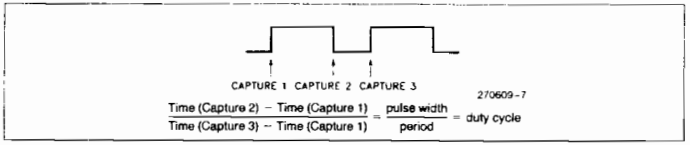


Figure 7. Measuring Duty Cycle

Listing 3. Measuring Duty Cycle

```
; RAM locations to store capture values
   CAPTURE            DATA      30H
   PULSE_WIDTH        DATA      32H
   PERIOD             DATA      34H
   FLAG_1             BIT       20H.0
   FLAG_2             BIT       20H.1
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:
; Initialization for PCA timer, module, and interrupt the same
; as in Listing 1. Capture positive edge first, then either
; edge.
;
;************************************************************
;                  Main program goes here                   ;
;************************************************************
;
; This code assumes only Module 0 is being used.
PCA_INTERRUPT:
   CLR CCF0                 ; Clear Module 0's event flag
   JB FLAG_1, SECOND_CAPTURE
;
FIRST_CAPTURE:
   MOV CAPTURE, CCAP0L
   MOV CAPTURE+1, CCAP0H
   SETB FLAG_1              ; Signify first capture complete
   MOV CCAPM0, #31H         ; Capture either edge now
   RETI
```

Listing 3. Measuring Duty Cycle (Continued)

```
;
SECOND_CAPTURE:
   PUSH ACC
   PUSH PSW
   JB FLAG_2, THIRD_CAPTURE
   CLR C                    ; Calculate pulse width
   MOV A, CCAP0L            ; 16-bit subtract
   SUBB A, CAPTURE
   MOV PULSE_WIDTH, A
   MOV A, CCAP0H
   SUBB A, CAPTURE+1
   MOV PULSE_WIDTH+1, A
   SETB FLAG_2              ; Signify second capture complete
   POP PSW
   POP ACC
   RETI
;
THIRD_CAPTURE:
   CLR C                    ; Calculate period
   MOV A, CCAP0L            ; 16-bit subtract
   SUBB A, CAPTURE
   MOV PERIOD, A
   MOV A, CCAP0H
   SUBB A, CAPTURE+1
   MOV PERIOD+1, A
   MOV CCAPM0, #21H         ; Optional - reconfigure module to
   CLR FLAG_1               ; capture positive edges for next
   CLR FLAG_2               ; cycle
   POP PSW
   POP ACC
   RETI
```

After the third capture, a 16-bit by 16-bit divide routine needs to be executed. This routine is located in Appendix B. Due to its length, it's up to the user whether the divide routine should be completed in the interrupt routine or be called as a subroutine from the main program.

Measuring Phase Differences

Because the PCA modules share the same time base, the PCA is useful for measuring the phase difference between two or more signals. For this example, two signals are input to Modules 0 and 1 as seen in Figure 8. Both modules are programmed to capture rising edges only. Listing 4 shows the code needed to measure the difference between these two signals. This code does not assume one signal is leading or lagging the other.

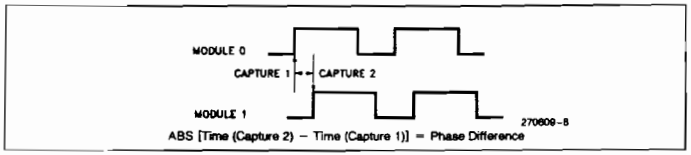


Figure 8. Measuring Phase Differences

Listing 4. Measuring Phase Differences

```
; RAM locations to store capture values
   CAPTURE_0            DATA     30H
   CAPTURE_1            DATA     32H
   PHASE                DATA     34H
   FLAG_0               BIT      20H.0
   FLAG_1               BIT      20H.1
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:
; Same initialization for PCA timer, and interrupt as
; in Listing 1. Initialize two PCA modules as follows:
;
   MOV CCAPM0, #21H         ; Module 0 capture rising edges
   MOV CCAPM1, #21H         ; Module 1 same
;
;*****************************************************************
;                  Main program goes here
;*****************************************************************
;
; This code assumes only Modules 0 and 1 are being used.
PCA_INTERRUPT:
   JB CCF0, MODULE_0        ; Determine which module's
   JB CCF1, MODULE_1        ; event caused the interrupt
;
MODULE_0:
   CLR CCF0                 ; Clear Module 0's event flag
   MOV CAPTURE_0, CCAP0L    ; Save 16-bit capture value
   MOV CAPTURE_0+1, CCAP0H
   JB FLAG_1, CALCULATE_PHASE ; If capture complete on
                            ; Module 1, go to calculation
   SETB FLAG_0              ; Signify capture on Module 0
   RETI
```

Listing 4. Measuring Phase Differences (Continued)

```
MODULE_1:
   CLR CCF_1                        ; Clear Module 1's event flag
   MOV CAPTURE_1, CCAP1L
   MOV CAPTURE_1+1, CCAP1H
   JB FLAG_0, CALCULATE_PHASE       ; If capture complete on
                                    ; Module 0, go to calculation
   SETB FLAG_1                      ; Signify capture on Module 1
   RETI
;
CALCULATE_PHASE:
   PUSH ACC                         ; This calculation does not
   PUSH PSW                         ; have to be completed in the
   CLR C                            ; interrupt service routine
;
   JB FLAG_0, MOD0_LEADING
   JB FLAG_1, MOD1_LEADING
;
MOD0_LEADING:
   MOV A, CAPTURE_1
   SUBB A, CAPTURE_0
   MOV PHASE, A
   MOV A, CAPTURE_1+1
   SUBB A, CAPTURE_0+1
   MOV PHASE+1, A
   CLR FLAG_0
   JMP EXIT
;
MOD1_LEADING:
   MOV A, CAPTURE_0
   SUBB A, CAPTURE_1
   MOV PHASE, A
   MOV A, CAPTURE_0+1
   SUBB A, CAPTURE_1+1
   MOV PHASE+1, A
   CLR FLAG_1
EXIT:
   POP PSW
   POP ACC
   RETI
```


Reading the PCA Timer

Some applications may require that the PCA timer be read instantaneously as a real-time event. Since the timer consists of two 8-bit registers (CH,CL), it would normally take two MOV instructions to read the whole timer. An invalid read could occur if the registers rolled over in the middle of the two MOVs.

However, with the capture mode a 16-bit timer value can be loaded into the capture registers by toggling a port pin. For example, configure Module 0 to capture falling edges and initialize P1.3 to be high. Then when the user wants to read the PCA timer, clear P1.3 and the full 16-bit timer value will be saved in the capture registers. It's still optional whether the user wants to generate an interrupt with the capture.

COMPARE MODE

In this mode, the 16-bit value of the PCA timer is compared with a 16-bit value pre-loaded in the module's compare registers. The comparison occurs three times per machine cycle in order to recognize the fastest possible clock input, i.e. $1/4$ x oscillator frequency. When there is a match, one of three events can happen:

(1) an interrupt — Software Timer mode
(2) toggle of a port pin — High Speed Output mode
(3) a reset — Watchdog Timer mode.

Examples of each compare mode will follow.

SOFTWARE TIMER

In most applications a software timer is used to trigger interrupt routines which must occur at periodic intervals. Figure 9 shows the sequence of events for the Software Timer mode. The user preloads a 16-bit value in a module's compare registers. When a match occurs between this compare value and the PCA timer, an event flag is set and an interrupt is flagged. An interrupt is then generated if it has been enabled.

If necessary, a new 16-bit compare value can be loaded into (CCAP0H, CCAP0L) during the interrupt routine. *The user should be aware that the hardware temporarily disables the comparator function while these registers are being updated so that an invalid match will not occur.* That is, a write to the low byte (CCAPn0) disables the comparator while a write to the high byte (CCAP0H) re-enables the comparator. For this reason, user software must write to CCAP0L first, then CCAP0H. The user may also want to hold off any interrupts from occurring while these registers are being updated. This can easily be done by clearing the EA bit. See the code example in Listing 5.

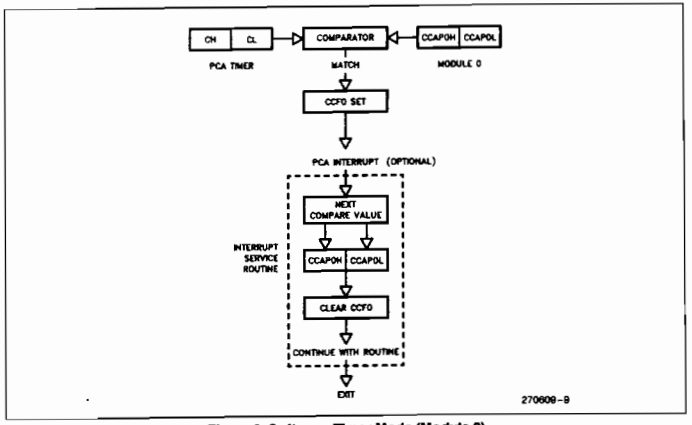


Figure 9. Software Timer Mode (Module 0)

Listing 5. Software Timer

```
; Generate an interrupt in software every 20 msec
;
;
; Frequency       = 12 MHz
; PCA clock input = 1/12 x Fosc  ⟶  1 µsec
;
; Calculate reload value for compare registers:
;              20 msec
;          ------------- = 20,000 counts
;          1 µsec/count
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
PCA_INIT:
; Initialize PCA timer same as in Listing 1
;  MOV CCAPM0, #49H          ; Module 0 in Software Timer mode
   MOV CCAPOL, #LOW(20000)   ; Write to low byte first
   MOV CCAPOH, #HIGH(20000)
;
   SETB EC                   ; Enable PCA interrupt
   SETB EA
   SETB CR                   ; Turn on PCA timer
;
;  ************************************************************
;                  Main program goes here
;  ************************************************************
;
PCA_INTERRUPT:
   CLR CCF0                  ; Clear Module 0's event flag
   PUSH ACC
   PUSH PSW
   CLR EA                    ; Hold off interrupts
   MOV A, #LOW(20000)        ; 16-Bit Add
   ADD A, CCAPOL             ; Next match will occur
   MOV CCAPOL, A             ; 20,000 counts later
   MOV A, #HIGH(20000)
   ADDC A, CCAPOH
   MOV CCAPOH, A
   SETB EA
;       .
;       .
; Continue with routine
;       .
;       .
   POP PSW
   POP ACC
   RETI
```


HIGH SPEED OUTPUT

The High Speed Output (HSO) mode toggles a port pin when a match occurs between the PCA timer and the preloaded value in the compare registers (see Figure 10). The HSO mode is more accurate than toggling pins in software because the toggle occurs *before* branching to an interrupt, i.e. interrupt latency will not effect the accuracy of the output. In fact, the interrupt is optional. Only if the user wants to change the time for the next toggle is it necessary to update the compare registers. Otherwise, the next toggle will occur when the PCA timer rolls over and matches the last compare value. Examples of both are shown.

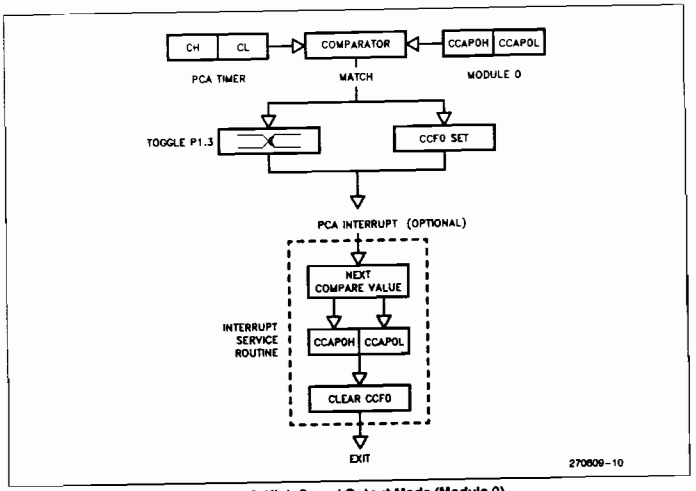


Figure 10. High Speed Output Mode (Module 0)

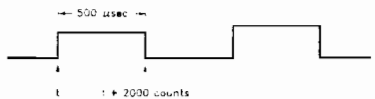
Without any CPU intervention, the fastest waveform the PCA can generate with the HSO mode is a 30.5 Hz signal at 16 MHz. Refer to Listing 6. By changing the PCA clock input, slower waveforms can also be generated.

Listing 6. High Speed Output (Without Interrupt)

```
; Maximum output with HSO mode without interrupts = 30.5 Hz signal
; Frequency       = 16 MHz
; PCA clock input = 1/4 x Fosc  ⟶  250 nsec
;
   MOV CMOD, #02H
   MOV CL, #00H
   MOV CH, #00H
   MOV CCAPM0, #4CH      ; HSO mode without interrupt enabled
   MOV CCAPOL, #0FFH     ; Write to low byte first
   MOV CCAPOH, #0FFH     ; P1.3 will toggle every 2^16 counts
                         ; or 16.4 msec
         .               ; Period = 30.5 Hz
   SETB CR               ; Turn on PCA timer
```

In this next example, the PCA interrupt is used to change the compare value for each toggle. This way a variable frequency output can be generated. Listing 7 shows an output of 1 KHz at 16 Mhz.

Listing 7. High Speed Output (With Interrupt)



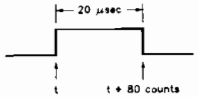
$$\frac{500\ \mu sec}{250\ nsec/count} = 2000\ counts$$

270609-11

```
   ORG 0000H
   JMP PCA_INIT
   ORG 0033H
   JMP PCA_INTERRUPT
;
PCA_INIT:
   MOV CMOD, #02H                ; Clock input = 250 nsec
   MOV CL, #00H                  ; at 16 MHz
   MOV CH, #00H
   MOV CCAPM0, #4DH              ; Module 0 in HSO mode with
   MOV CCAP0L, #LOW(1000)        ; PCA interrupt enabled
   MOV CCAP0H, #HIGH(1000)       ; t = 1000 (arbitrary)
   CLR P1.3
;
   SETB EC                       ; Enable PCA interrupt
   SETB EA
   SETB CR                       ; Turn on PCA timer
;
;  ***************************************************************
;                  Main program goes here
;  ***************************************************************
;
; This code assumes only Module 0 is being used.
PCA_INTERRUPT:
   CLR CCF0                      ; Clear Module 0's event flag
   PUSH ACC
   PUSH PSW
   CLR EA                        ; Hold off interrupts
   MOV A, #LOW(2000)             ; 16-bit add
   ADD A, CCAP0L                 ; 2000 counts later, P1.3
   MOV CCAP0L, A                 ; will toggle
   MOV A, #HIGH(2000)
   ADDC A, CCAP0H
   MOV CCAP0H, A
   SETB EA
   POP PSW
   POP ACC
   RETI
```


Another option with the HSO mode is to generate a single pulse. Listing 8 shows the code for an output with a pulse width of 20 μsec. As in the previous example, the PCA interrupt will be used to change the time for the toggle. The first toggle will occur at time "t". After 80 counts of the PCA timer, 20 μsec will have expired, and the next toggle will occur. Then the HSO mode will be disabled.

Listing 8. High Speed Output (Single Pulse)



$$\frac{20\ \mu sec}{250\ nsec/count} = 80\ counts$$

270609-12

```
   ORG 0000H
   JMP PCA_INIT
   ORG 0033H
   JMP PCA_INTERRUPT
;
PCA_INIT:
   MOV CMOD, #02H                ; Clock input = 250 nsec
   MOV CL, #00H                  ; at 16 MHz
   MOV CH, #00H
   MOV CCAPM0, #4DH              ; Module 0 in HSO mode with
   MOV CCAP0L, #LOW(1000)        ; PCA interrupt enabled
   MOV CCAP0H, #HIGH(1000)       ; t = 1000 (arbitrary)
   CLR P1.3
   :
   SETB EC                       ; Enable PCA interrupt
   SETB EA
   SETB CR                       ; Turn on PCA timer
;
;  ***************************************************************
;                  Main program goes here
;  ***************************************************************
;
; This code assumes only Module 0 is being used.
PCA_INTERRUPT:
   CLR CCF0                      ; Clear Module 0's event flag
   JNB P1.3, DONE
;
   PUSH ACC
   PUSH PSW
   CLR EA                        ; Hold off interrupts
   MOV A, #LOW(80)               ; 16-bit add
   ADD A, CCAP0L                 ; 80 counts later, P1.3
   MOV CCAP0L, A                 ; will toggle
   MOV A, #HIGH(80)
   ADDC A, CCAP0H
   MOV CCAP0H, A
   SETB EA
   POP PSW
   POP ACC
   RETI
;
DONE:
   MOV CCAPM0, #00H              ; Disable HSO mode
   RETI
```

WATCHDOG TIMER

An on-board watchdog timer is available with the PCA to improve the reliability of the system without increasing chip count. Watchdog timers are useful for systems which are susceptible to noise, power glitches, or electrostatic discharge. Module 4 is the only PCA module which can be programmed as a watchdog. However, this module can still be used for other modes if the watchdog is not needed.

Figure 11 shows a diagram of how the watchdog works. The user pre-loads a 16-bit value in the compare registers. Just like the other compare modes, this 16-bit value is compared to the PCA timer value. If a match is allowed to occur, an internal reset will be generated. This will not cause the RST pin to be driven high.

In order to hold off the reset, the user has three options:

(1) periodically change the compare value so it will never match the PCA timer,

(2) periodically change the PCA timer value so it will never match the compare value, or

(3) disable the watchdog by clearing the WDTE bit before a match occurs and then re-enable it.

The first two options are more reliable because the watchdog timer is never disabled as in option #3. If the program counter ever goes astray, a match will eventually occur and cause an internal reset. The second option is also not recommended if other PCA modules are being used. Remember, the PCA timer is the time base for *all* modules; changing the time base for other modules would not be a good idea. Thus, in most applications the first solution is the best option.

Listing 9 shows the code for initializing the watchdog timer. Module 4 can be configured in either compare mode, and the WDTE bit in CMOD must also be set. The user's software then must periodically change (CCAP4H,CCAP4L) to keep a match from occurring with the PCA timer (CH,CL). This code is given in the WATCHDOG routine.

This routine should not be part of an interrupt service routine. Why? Because if the program counter goes astray and gets stuck in an infinite loop, interrupts will still be serviced and the watchdog will keep getting reset. Thus, the purpose of the watchdog would be defeated. Instead call this subroutine from the main program within 2^{16} count of the PCA timer.

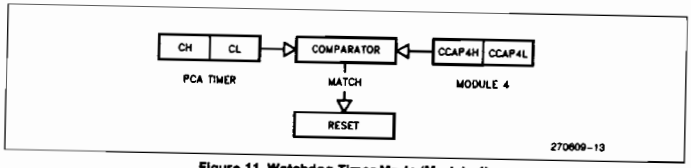


Figure 11. Watchdog Timer Mode (Module 4)

Listing 9. Watchdog Timer

```
INIT_WATCHDOG:
   MOV CCAPM4, #4CH          ; Module 4 in compare mode
   MOV CCAP4L, #0FFH         ; Write to low byte first
   MOV CCAP4H, #0FFH         ; Before PCA timer counts up to
                             ; FFFF Hex, these compare values
                             ; must be changed
   ORL CMOD, #40H            ; Set the WDTE bit to enable the
                             ; watchdog timer without changing
                             ; the other bits in CMOD
;
;*************************************************************************
;
; Main program goes here, but CALL WATCHDOG periodically.
;
;*************************************************************************
;
WATCHDOG:
   CLR EA                    ; Hold off interrupts
   MOV CCAP4L, #00           ; Next compare value is within
   MOV CCAP4H, CH            ; 255 counts of the current PCA
   SETB EA                   ; timer value
   RET
```

PULSE WIDTH MODULATOR

The PCA can generate 8-bit PWMs by comparing the low byte of the PCA timer (CL) with the low byte of the compare registers (CCAPnL). When CL < CCAPnL the output is low. When CL ≥ CCAPnL the output is high.

To control the duty cycle of the output, the user actually loads a value into the high byte CCAPnH (see Figure 12). Since a write to this register is asynchronous, a new value is not shifted into CCAPnL for comparison until the next period of the output: that is, when CL rolls over from 255 to 00. This mechanism provides "glitch-free" writes to CCAPnH when the duty cycle of the output is changed.

CCAPnH can contain any integer from 0 to 255, but Figure 13 shows a few common duty cycles and the corresponding values for CCAPnH. Note that a 0% duty cycle can be obtained by writing to the port pin directly with the CLR bit instruction. To calculate the CCAPnH value for a given duty cycle, use the following equation:

$$CCAPnH = 256\ (1 - \text{Duty Cycle})$$

where CCAPnH is an 8-bit integer and Duty Cycle is expressed as a fraction.

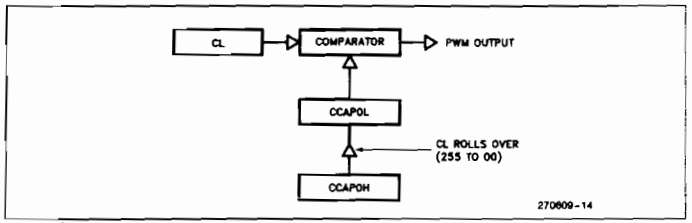


Figure 12. PWM Mode (Module 0)

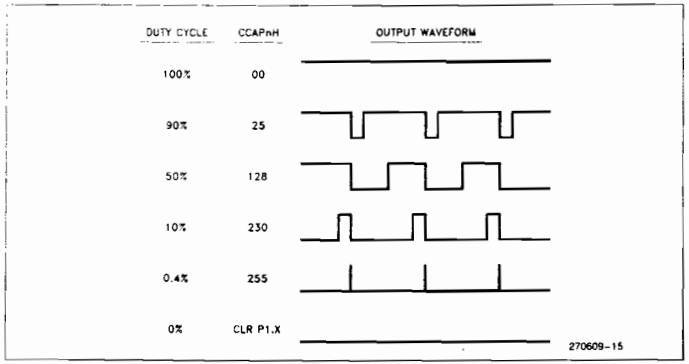


Figure 13. CCAPnH Varies Duty Cycle

Table 4. PWM Frequencies.

PCA Timer Mode	PWM Frequency	
	12 MHz	16 MHz
1/12 Osc. Frequency	3.9 KHz	5.2 KHz
¼ Osc. Frequency	11.8 KHz	15.6 KHz
Timer 0 Overflow:		
8-bit	15.5 Hz	20.3 Hz
16-bit	0.06 Hz	0.08 Hz
8-bit Auto-Reload	3.9 KHz to 15.3 Hz	5.2 KHz to 20.3 Hz
External Input (Max)	5.9 KHz	7.8 KHz

Listing 10. PWM

```
INIT-PWM:
   MOV CMOD, #02H          ; Clock input = 250 nsec at 16 MHz
   MOV CL, #00H            ; Frequency of output = 15.6 KHz
   MOV CH, #00H
   MOV CCAPM0, #42H        ; Module 0 in PWM mode
   MOV CCAP0L, #00H
   MOV CCAP0H, #128D       ; 50 percent duty cycle
;
   SETB CR                 ; Turn on PCA timer
```

The frequency of the PWM output will depend on which of the four inputs is chosen for the PCA timer. The maximum frequency is 15.6 KHz at 16 MHz. Refer to Table 4 for a summary of the different PWM frequencies possible with the PCA.

Listing 10 shows how to initialize Module 0 for a PWM signal at 50% duty cycle. Notice that no PCA interrupt is needed to generate the PWM (i.e no software overhead!). To create a PWM output on the 8051 requires a hardware timer plus software overhead to toggle the port pin. The advantage of the PCA is obvious, not to mention it can support up to 5 PWM outputs with just one chip.

CONCLUSION

This list of examples with the PCA is by no means exhaustive. However, the advantages of the PCA can easily be seen from the given applications. For example, the PCA can provide better resolution than Timers 0, 1 and 2 because the PCA clock rate can be three times faster. The PCA can also perform many tasks that these hardware timers can not, i.e. measure phase differences between signals or generate PWMs. In a sense, the PCA provides the user with five more timer/counters in addition to Timers 0, 1 and 2 on the 8XC51FA/FB.

Appendix A includes test routines for all the software examples in this application note. The divide routine for calculating duty cycles is in Appendix B. And finally, Appendix C is a table of the Special Function Registers for the 8XC51FA/FB with the new or modified registers **boldfaced**.

APPENDIX A
TEST ROUTINES

```
;                Listing 1a - Measuring Pulse Widths

$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list
;
;       Variables
;
CAPTURE                 DATA    30H
PULSE_WIDTH             DATA    32H
FLAG                    BIT     20H.0
;
ORG 0000H
JMP PCA_INIT
;
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #00H                  ; Input to PCA timer = 1/12 x Fosc
                MOV CH, #00
                MOV CL, #00
;
;       Initialize Module 0 in capture mode
                MOV CCAPM0, #21H                ; Capture positive edge first on P1.3

                MOV CCAP0H, #00
                MOV CCAP0L, #00
;
                SETB EC                         ; Enable PCA interrupt
                SETB EA
                SETB CR                         ; Turn PCA timer on
                CLR FLAG                        ; Clear test flag
;
;*****************************************************************************
;                       Test program only
WAIT:           JMP $                           ; Wait for PCA interrupt
                JMP WAIT
;*****************************************************************************
;
;       This code assumes Module 0 is the only module being used.  If
;       other PCA module's are being used, software must check which
;       module's event flag caused the interrupt.
;
PCA_INTERRUPT:
                CLR CCF0                        ; Clear module 0's event flag
                JB FLAG, SECOND_CAPTURE
;
FIRST_CAPTURE:
                MOV CAPTURE, CCAP0L
                MOV CAPTURE+1, CCAP0H
```

270609-16


```
                MOV CCAPM0, #11H                ; Change module to now capture
                                                ; falling edges
                SETB FLAG                       ; Signify first capture complete
                RETI
;
SECOND_CAPTURE:
                PUSH ACC
                PUSH PSW
                CLR C
                MOV A, CCAP0L                   ; 16-bit subtract
                SUBB A, CAPTURE
                MOV PULSE_WIDTH, A
                MOV A, CCAP0H
                SUBB A, CAPTURE+1
                MOV PULSE_WIDTH+1, A
;
                MOV CCAPM0, #21H                ; Optional if user wants to measure
                CLR FLAG                        ; next pulse width
                POP PSW
                POP ACC
                RETI
;
END
```

270609-17

Listing 1b - Measuring Periods

```
;
$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list
;
;       Variables
;
CAPTURE         DATA    30H
PERIOD          DATA    32H
FLAG            BIT     20H.0
;
ORG 0000H
JMP PCA_INIT
;
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #00H          ; Input to timer = 1/12 x Fosc
                MOV CH, #00H
                MOV CL, #00
;
;       Initialize Module 0 in capture mode
                MOV CCAPM0, #21H        ; Capture rising edges on P1.3

                MOV CCAP0H, #00
                MOV CCAP0L, #00
;
                SETB EC                 ; Enable PCA interrupt
                SETB EA
                SETB CR                 ; Turn PCA timer on
                CLR FLAG                ; Clear test flag
;
;*****************************************************************************
;               Test program only
;
WAIT:           JMP $                   ; Wait for PCA interrupt
                JMP WAIT
;*****************************************************************************
;
;       This code assumes only Module 0 is being used. If other modules
;       are being used, software must check which module's flag caused
;       the interrupt.
;
PCA_INTERRUPT:
                CLR CCF0                ; Clear module 0's event flag
                JB FLAG, SECOND_CAPTURE
;
FIRST_CAPTURE:
                MOV CAPTURE, CCAP0L
                MOV CAPTURE+1, CCAP0H
```

270609-18


```
                SETB FLAG               ; Signify first capture complete
                RETI
;
SECOND_CAPTURE:
                PUSH ACC
                PUSH PSW
                CLR C
                MOV A, CCAP0L           ; 16-Bit subtraction
                SUBB A, CAPTURE
                MOV PERIOD, A
                MOV A, CCAP0H
                SUBB A, CAPTURE+1
                MOV PERIOD+1, A
;
                CLR FLAG
                POP PSW
                POP ACC
                RETI
;
END
```

270609-19

Listing 2: Measuring Frequencies

```
$nomod51
$nosymbols
$nolist
$include (reg352.sdl)
$list

        Variables

CAPTURE                 DATA    30H
PERIOD                  DATA    32H
SAMPLE_COUNT            DATA    34H
FLAG                    BIT     20H.0

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

;       Initialize PCA timer
PCA_INIT:   MOV CMOD, #00H              ; Input to PCA timer = 1/12 x Fosc
            MOV CH, #00
            MOV CL, #00

;       Initialize Module 0 in capture mode
            MOV CCAPM0, #21H            ; Capture positive edges on P1.3

            MOV CCAP0H, #00
            MOV CCAP0L, #00

            MOV SAMPLE_COUNT, #10D      ; N = 10 for this example

            SETB EC                     ; Enable PCA interrupt
            SETB EA
            SETB CR                     ; Turn PCA timer on
            CLR FLAG                    ; Test flag
;*****************************************************************************
;               Test program only
;
WAIT:       JMP $                       ; Wait for PCA interrupt
            JMP WAIT
;*****************************************************************************
;
;       This code assumes only Module 0 is being used.
;
PCA_INTERRUPT:
            CLR CCF0                    ; Clear module 0's event flag
            JB FLAG, NEXT_CAPTURE
;
```

270609-20


```
FIRST_CAPTURE:
            MOV CAPTURE, CCAP0L
            MOV CAPTURE+1, CCAP0H
            SETB FLAG                   ; Signify first capture complete
            RETI
;
NEXT_CAPTURE:
            DJNZ SAMPLE_COUNT, EXIT
            PUSH ACC
            PUSH PSW
            CLR C
            MOV A, CCAP0L               ; 16-Bit subtraction
            SUBB A, CAPTURE
            MOV PERIOD, A
            MOV A, CCAP0H
            SUBB A, CAPTURE+1
            MOV PERIOD+1, A
;
            MOV SAMPLE_COUNT, #10D      ; Reload for next capture
            CLR FLAG
            POP PSW
            POP ACC
EXIT:       RETI
;
END
```

270609-21

Listing 3 - Measuring Duty Cycle

```
;
$nomod51
$nosymbols
$noilist
$include (reg252.pdf)
$list
;
        Variables
;
CAPTURE                 DATA    30H
PULSE_WIDTH             DATA    32H
PERIOD                  DATA    34H
;
FLAG_1                  BIT     20H.0
FLAG_2                  BIT     20H.1
;
ORG 0000H
JMP PCA_INIT
;
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #00H              ; Input to PCA timer = 1/12 x Fosc
                MOV CH, #00
                MOV CL, #00
;
;       Initialize Module 0 in capture mode
                MOV CCAPM0, #21H            ; Capture positive edge first on P1.3
                MOV CCAP0H, #00
                MOV CCAP0L, #00
;
                CLR FLAG_1                  ; Clear test flags
                CLR FLAG_2
;
                SETB EC                     ; Enable PCA interrupt
                SETB EA
                SETB CR                     ; Turn PCA timer on
;
;*****************************************************************************
;               Test program only
;
WAIT:           JMP $                       ; Wait for PCA interrupt
                JMP WAIT
;*****************************************************************************
;
;       This code assumes Module 0 is the only PCA module being used.
;
PCA_INTERRUPT:
                CLR CCF0                    ; Clear module 0's event flag
                JB FLAG_1, SECOND_CAPTURE
;
```

270609-22


```
FIRST_CAPTURE:
                MOV CAPTURE, CCAP0L
                MOV CAPTURE+1, CCAP0H
                SETB FLAG_1                 ; Signify first capture complete
                MOV CCAPM0, #31H            ; Capture either edge now
                RETI
;
SECOND_CAPTURE:
                PUSH ACC
                PUSH PSW
                JB FLAG_2, THIRD_CAPTURE
                CLR C                       ; Calculate pulse width
                MOV A, CCAP0L               ; 16-bit subtract
                SUBB A, CAPTURE
                MOV PULSE_WIDTH, A
                MOV A, CCAP0H
                SUBB A, CAPTURE+1
                MOV PULSE_WIDTH+1, A
;
                SETB FLAG_2                 ; Signify second capture complete
                POP PSW
                POP ACC
                RETI
;
THIRD_CAPTURE:
                CLR C                       ; Calculate period
                MOV A, CCAP0L               ; 16-bit subtract
                SUBB A, CAPTURE
                MOV PERIOD, A
                MOV A, CCAP0H
                SUBB A, CAPTURE+1
                MOV PERIOD+1, A
;
                MOV CCAPM0, #21H            ; Optional- reconfigure module to
                CLR FLAG_1                  ; capture positive edges for
                CLR FLAG_2                  ; next cycle
                POP PSW
                POP ACC
                RETI
;
END
```

270609-23

Listing 4 - Measuring Phase Differences

```
;
$nomod51
$nosymbols
$noflist
$include (reg252.pdf)
$list
;
        Variables
;
CAPTURE_0               DATA    30H
CAPTURE_1               DATA    32H
PHASE                   DATA    34H
;
FLAG_0                  BIT     20H.0
FLAG_1                  BIT     20H.1
;
ORG 0000H
JMP PCA_INIT
;
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #00H          ; Input to PCA timer = 1/12 x Fosc
                MOV CH, #00
                MOV CL, #00
;
;       Initialize Modules 0 & 1 in capture mode
                MOV CCAPM0, #21H        ; Capture positive edges on P1.3

                MOV CCAP0H, #00
                MOV CCAP0L, #00
;
                MOV CCAPM1, #21H        ; Capture positive edges on P1.4
                MOV CCAP1H, #00
                MOV CCAP1L, #00
;
                MOV R0, #0FFH           ; Used for test program only
                MOV R1, #0FFH
;
                CLR FLAG_0              ; Clear test flags
                CLR FLAG_1
;
                SETB EC                 ; Enable PCA interrupt
                SETB EA
                SETB CR                 ; Turn PCA timer on
;
```

270609-24


```
;.....
;               Test program only
;
MAIN:           CALL TOG1               ; Generate two waveforms
                CALL DELAY2             ; with known phase difference
                CALL TOG2
                JMP MAIN
;
TOG1:           CPL P1.6                ; These two waveforms are input to
                CALL DELAY1             ; P1.3 and P1.4
                RET
;
TOG2:           CPL P1.5
                CALL DELAY1
                RET
;
DELAY1:         DJNZ R0, $
                RET
;
DELAY2:         DJNZ R1, $
                RET
;.....
;
;       This code assumes only Modules 0 and 1 are being used.
;
PCA_INTERRUPT:
                JB CCF0, MODULE_0       ; Determine which module's event
                JB CCF1, MODULE_1       ; caused the interrupt
;
MODULE_0:
                CLR CCF0                ; Clear Module 0's event flag
                MOV CAPTURE_0, CCAP0L
                MOV CAPTURE_0+1, CCAP0H
                JB FLAG_1, CALCULATE_PHASE ; If capture is complete on Module 1,
                                        ; go to calculation
                SETB FLAG_0             ; Signify capture complete on
                RETI                    ; Module 0
;
MODULE_1:
                CLR CCF1                ; Clear Module 1's event flag
                MOV CAPTURE_1, CCAP1L
                MOV CAPTURE_1+1, CCAP1H
                JB FLAG_0, CALCULATE_PHASE ; If capture is complete on Module 0,
                                        ; go to calculation
                SETB FLAG_1             ; Signify capture complete
                RETI                    ; Module 1
;
CALCULATE_PHASE:
                PUSH ACC                ; This calculation does not have to
                PUSH PSW                ; be completed in the interrupt
                CLR C                   ; service routine
;
                JB FLAG_0, MOD0_LEADING
```

270609-25

```
        JB FLAG_1, MOD1_LEADING
;
MOD0_LEADING:
        MOV A, CAPTURE_1            ; 16-bit subtraction
        SUBB A, CAPTURE_0
        MOV PHASE, A
        MOV A, CAPTURE_1+1
        SUBB A, CAPTURE_0+1
        MOV PHASE+1, A
        CLR FLAG_0
        JMP EXIT
;
MOD1_LEADING:
        MOV A, CAPTURE_0            ; 16-bit subtraction
        SUBB A, CAPTURE_1
        MOV PHASE, A
        MOV A, CAPTURE_0+1
        SUBB A, CAPTURE_1+1
        MOV PHASE+1, A
        CLR FLAG_1
EXIT:
        POP PSW
        POP ACC
        RETI
;
END
```

270609-26

Listing 5. Software Timer

```
;
$nomod51
$nosymbols
$noIist
$include (reg252.pdf)
$list
;       Software Timer mode which interrupts every 20 msec with Fosc = 12 MHz
;
ORG 0000H
JMP PCA_INIT
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT    MOV CMOD, #00H              ; Input to PCA timer = 1:12 x Fosc
            MOV CH, #00
            MOV CL, #00
;
            MOV CCAPM0, #49H            ; Software Timer mode with interrupt
            MOV CCAP0L, #LOW(20000)     ; Write to low byte first
            MOV CCAP0H, #HIGH(20000)
;
            SETB EC                     ; Enable PCA interrupt
            SETB EA
            SETB CR                     ; Turn PCA timer on
;
;*****************************************************************************
;
;           Test program only
;
WAIT:       JMP $                       ; Wait for PCA interrupt
            JMP WAIT
;*****************************************************************************
;
;       This code assumes Module 0 is the only module being used.  If
;       other PCA module's are being used, software must check which
;       module's event flag caused the interrupt.
;
PCA_INTERRUPT:
            CLR CCF0                    ; Clear module 0's event flag
            PUSH ACC
            PUSH PSW
            CLR EA                      ; Hold off interrupts
            MOV A, #LOW(20000)          ; 16-bit add
            ADD A, CCAP0L               ; Next match will occur 20,000
            MOV CCAP0L, A               ; counts later
            MOV A, #HIGH(20000)
            ADDC A, CCAP0H
            MOV CCAP0H, A
            SETB EA
            POP PSW
            POP ACC
            RETI
END
```

270609-27

Listing 6. High Speed Output (without interrupt)

```
$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list

;       HSO mode without PCA interrupt.  Maximum frequency output = 30.5 Hz
;       at Fosc = 16 MHz.

ORG 0000H
JMP PCA_INIT

;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #02H          ;Input to PCA timer = 1/4 x Fosc
                MOV CH, #00
                MOV CL, #00
;
                MOV CCAPM0, #4CH        ; HSO Mode without interrupt enabled
                MOV CCAP0L, #0FFH       ; Write to low byte first
                MOV CCAP0H, #0FFH       ; P1.3 will toggle every 65,536 counts
                                        ; or 16.4 msec at Fosc = 16 MHz
                                        ; Period = 30.5 Hz
                SETB CR                 ; Turn PCA timer on
;
END
```

270609-28

Listing 7. High Speed Output (with interrupts)

```
$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list

;       HSO mode with variable frequency.  This example outputs a 1KHz signal
;       with Fosc = 16 MHz.

ORG 0000H
JMP PCA_INIT

ORG 0033H
JMP PCA_INTERRUPT

;       Initialize PCA timer
PCA_INIT:       MOV CMOD, #02H          ; Input to PCA timer = 1/4 x Fosc
                MOV CH, #00
                MOV CL, #00
;
                MOV CCAPM0, #4DH        ; HSO mode with interrupt enabled
                MOV CCAP0L, #LOW(1000)  ; t = 1000 arbitrary
                MOV CCAP0H, #HIGH(1000)
                CLR P1.3
;
                SETB EC                 ; Enable PCA interrupt
                SETB EA
                SETB CR                 ; Turn PCA timer on
;***************************************************************************
;               Test program only
;
WAIT:           JMP $                   ; Wait for PCA interrupt
                JMP WAIT
;***************************************************************************
;       This code assumes Module 0 is the only module being used.  If
;       other PCA module's are being used, software must check which
;       module's event flag caused the interrupt.
;
PCA_INTERRUPT:
                CLR CCF0                ; Clear module 0's event flag
                PUSH ACC
                PUSH PSW
                CLR EA                  ; Hold off interrupts
                MOV A, #LOW(2000)       ; 16-bit add
                ADD A, CCAP0L           ; 2000 counts later P1.3
                MOV CCAP0L, A           ; will toggle
                MOV A, #HIGH(2000)
                ADDC A, CCAP0H
                MOV CCAP0H, A
;
                SETB EA
                POP PSW
                POP ACC
                RETI
;
END
```

270609-29

270609-30

Listing 8. High Speed Output (Single Pulse)

```
$nomod51
$nosymbols
$noilist
$include (reg252.pdf)
$list
;
;       HSO mode generates a single pulse width of 20 usecs with Fosc = 16 MHz.
;
ORG 0000H
JMP PCA_INIT
;
ORG 0033H
JMP PCA_INTERRUPT
;
;       Initialize PCA timer
PCA_INIT:   MOV CMOD, #02H          ; Input to PCA timer = 1/4 x Fosc
            MOV CH, #00
            MOV CL, #00
;
            MOV CCAPM0, #4DH        ; HSO mode with interrupt enabled
            MOV CCAP0L, #LOW(1000)  ; 1 = 1000 arbitrary
            MOV CCAP0H, #HIGH(1000)
            CLR P1.3
;
            SETB EC                 ; Enable PCA interrupt
            SETB EA
            SETB CR                 ; Turn PCA timer on
;
;*****************************************************************************
;           Test program only
;
WAIT:       JMP $                   ; Wait for PCA interrupt
            JMP WAIT
;*****************************************************************************
;
;       This code assumes Module 0 is the only module being used.  If
;       other PCA module's are being used, software must check which
;       module's event flag caused the interrupt.
;
PCA_INTERRUPT:
            CLR CCF0                ; Clear module 0's event flag
            JNB P1.3, DONE
;
            PUSH ACC
            PUSH PSW
            CLR EA                  ; Hold off interrupts
            MOV A, #LOW(80)         ; 16-bit add
            ADD A, CCAP0L           ; 80 counts later P1.3
            MOV CCAP0L, A           ; will toggle
            MOV A, #HIGH(80)
```



```
            ADDC A, CCAP0H
            MOV CCAP0H, A
            SETB EA
            POP PSW
            POP ACC
            RETI
;
DONE:
            MOV CCAPM0, #00H        ; Disable HSO mode
            RETI
END
```


Listing 9. Watchdog Timer

```
$nomod51
$nosymbols
$noilist
$include (reg252.pdf)
$list
;
ORG 0000H
JMP PCA_INIT
;
;       Initialize PCA timer
PCA_INIT:   MOV CMOD, #00H          ; Input to PCA timer = 1/12 x Fosc
            MOV CH, #00
            MOV CL, #00
;
            MOV CCAPM4, #4CH        ; Module 4 in compare mode
            MOV CCAP4L, #0FFH       ; Write to low byte first
            MOV CCAP4H, #0FFH       ; Before PCA timer counts up to FFFF Hex,
                                    ; these compare values must be changed
            ORL CMOD, #40H          ; Set the WDTE bit to enable watchdog timer
;
            SETB CR                 ; Turn PCA timer on
;
;*****************************************************************************
;           Test program only
;
START:      MOV R1, #120D           ; Delay for approx. 60 msec
            MOV R0, #0FFH
;
MAIN:       DJNZ R0, $              ; Check that watchdog never causes a reset
            DJNZ R1, MAIN
            CALL WATCHDOG
            JMP START
;
;*****************************************************************************
;
WATCHDOG:
            CLR EA                  ; Hold off interrupts
            MOV CCAP4L, #00H        ; Next compare value is within
            MOV CCAP4H, CH          ; 255 counts of the current PCA
            SETB EA                 ; timer value
            RET
;
END
```

Listing 10. Pulse Width Modulator

```
$nomod51
$nosymbols
$nolist
$include (reg252.pdf)
$list

        PWM mode – Maximum frequency output = 15.6 KHz with Fosc = 16 Mhz.

ORG 0000H
JMP PCA_INIT

        Initialize PCA timer
PCA_INIT:   MOV CMOD, #02H          ; Input to PCA timer = 1/4 x Fosc
            MOV CH, #00             ; At 16 MHz, frequency = 15.6 KHz
            MOV CL, #00

            MOV CCAPM0, #42H        ; PWM Mode
            MOV CCAP0L, #00H        ; Write to low byte first
            MOV CCAP0H, #128D       ; 50 percent duty cycle
            SETB CR                 ; Turn PCA timer on

END
```

270609-34

APPENDIX B
Duty Cycle Calculation

```
$DEBUG

SHORT_DIVISION SEGMENT CODE

EXTRN DATA(PULSE_WIDTH, PERIOD, DUTY_CYCLE)
PUBLIC DUTY_CYCLE_CALCULATION

RSEG SHORT_DIVISION

;*******************************************************************************
;
;           DUTY_CYCLE_CALCULATION
;
;   CALCULATES DUTY_CYCLE = PULSE_WIDTH / PERIOD
;
; Inputs to this routine are 16-bit pulse width and period measurements of
; a rectangular waveform. The output is a 9-bit BCD number representing
; the duty cycle of the waveform. The low 8 bits of the result are
; returned in DUTY_CYCLE. The 9th bit is the carry bit in the PSW. If the
; duty cycle is between 0 and 99 percent, the carry bit is 0 and DUTY_CYCLE
; contains the two BCD digits representing the duty cycle as a percent.
; If the duty cycle is 100 percent, the carry bit is 1 and DUTY_CYCLE
; contains 0.
;
; INPUTS: PULSE_WIDTH   2 bytes in externally defined DATA
;         (low byte at PULSE_WIDTH, high byte at PULSE_WIDTH+1)
;
;         PERIOD        2 bytes in externally defined DATA
;         (low byte at PERIOD, high byte at PERIOD+1)
;
; OUTPUT: DUTY_CYCLE    1 byte in externally defined DATA
;
; VARIABLES AND REGISTERS MODIFIED:
;
;     PULSE_WIDTH, DUTY_CYCLE
;     ACC, B, PSW, R2, R3
;
; ERROR EXIT:  Exit with OV = 1 indicates PULSE_WIDTH > PERIOD.
;
;*******************************************************************************


DUTY_CYCLE_CALCULATION:
        MOV   A,PERIOD+1
        CJNE  A,PULSE_WIDTH+1,NOT_EQUAL
        MOV   A,PERIOD
        CJNE  A,PULSE_WIDTH,NOT_EQUAL
```

270609-35

```
EQUAL:
        SETB  C
        MOV   DUTY_CYCLE,#0
        CLR   OV
        RET

NOT_EQUAL:
        JNC   CONTINUE
        SETB  OV
        RET

CONTINUE:
        MOV   R2,#8
        MOV   DUTY_CYCLE,#0
        MOV   R3,#0

TIMES_TWO:
        MOV   A,PULSE_WIDTH
        RLC   A
        MOV   PULSE_WIDTH,A
        MOV   A,PULSE_WIDTH+1
        RLC   A
        MOV   PULSE_WIDTH+1,A
        MOV   A,R3
        RLC   A
        MOV   R3,A
COMPARE:
        CJNE  R3,#0,DONE
        MOV   A,PULSE_WIDTH+1
        CJNE  A,PERIOD+1,DONE
        MOV   A,PULSE_WIDTH
        CJNE  A,PERIOD,DONE
DONE:
        CPL   C

BUILD_DUTY_CYCLE:
        MOV   A,DUTY_CYCLE
        RLC   A
        MOV   DUTY_CYCLE,A
        JNB   ACC.0,LOOP_CONTROL
SUBTRACT:
        MOV   A,PULSE_WIDTH
        SUBB  A,PERIOD
        MOV   PULSE_WIDTH,A
        MOV   A,PULSE_WIDTH+1
        SUBB  A,PERIOD+1
        MOV   PULSE_WIDTH+1,A
        MOV   A,R3
        SUBB  A,#0
        MOV   R3,A
LOOP_CONTROL:
        DJNZ  R2,TIMES_TWO
```

270609-36


```
FINAL_TIMES_TWO:
        MOV   A,PULSE_WIDTH
        RLC   A
        MOV   PULSE_WIDTH,A
        MOV   A,PULSE_WIDTH+1
        RLC   A
        MOV   PULSE_WIDTH+1,A
        MOV   A,R3
        RLC   A
        MOV   R3,A
FINAL_COMPARE:
        CJNE  R3,#0,FINAL_DONE
        MOV   A,PULSE_WIDTH+1
        CJNE  A,PERIOD+1,FINAL_DONE
        MOV   A,PULSE_WIDTH
        CJNE  A,PERIOD,FINAL_DONE
FINAL_DONE:
        JC    CONVERT_TO_BCD
        MOV   A,DUTY_CYCLE
        ADD   A,#1
        MOV   DUTY_CYCLE,A
        JNC   CONVERT_TO_BCD
        CLR   OV
        RET

CONVERT_TO_BCD:
        MOV   A,DUTY_CYCLE
        MOV   B,#10
        MUL   AB
        XCH   A,B
        SWAP  A
        MOV   DUTY_CYCLE,A
        MOV   A,#10
        MUL   AB
        XCH   A,B
        ORL   DUTY_CYCLE,A
        MOV   A,#10
        MUL   AB
        MOV   A,B
        CJNE  A,#5,TEST
TEST:   JBC   CY,OUT
        MOV   A,DUTY_CYCLE
        ADD   A,#1
        DA    A
        MOV   DUTY_CYCLE,A
OUT:    RET

END
```

270609-37

APPENDIX C

A map of the Special Function Register (SFR) space is shown in Table A1. Those registers which are new or have new bits added for the 83C51FA and 83C51FB have been **boldfaced**.

Note that not all of the addresses are occupied. Unoccupied addresses are not implemented on the chip. Read accesses to these addresses will in general return random data, and write accesses will have no effect.

User software should not write 1s to these unimplemented locations, since they may be used in future 8051 family products to invoke new features. In that case the reset or inactive values of the new bits will always be 0, and their active values will be 1.

Table A1. Special Function Register Memory Map and Values After Reset

F8		**CH 00000000**	**CCAP0H XXXXXXXX**	**CCAP1H XXXXXXXX**	**CCAP2H XXXXXXXX**	**CCAP3H XXXXXXXX**	**CCAP4H XXXXXXXX**		FF
F0	* B 00000000								F7
E8		**CL 00000000**	**CCAP0L XXXXXXXX**	**CCAP1L XXXXXXXX**	**CCAP2L XXXXXXXX**	**CCAP3L XXXXXXXX**	**CCAP4L XXXXXXXX**		EF
E0	* ACC 00000000								E7
D8	**CCON 00X00000**	**CMOD 00XXX000**	**CCAPM0 X0000000**	**CCAPM1 X0000000**	**CCAPM2 X0000000**	**CCAPM3 X0000000**	**CCAPM4 X0000000**		DF
D0	* PSW 00000000								D7
C8	T2CON 00000000	**T2MOD XXXXXX0**	RCAP2L 00000000	RCAP2H 00000000	TL2 00000000	TH2 00000000			CF
C0									C7
B8	*** IP X0000000**	**SADEN 00000000**							BF
B0	*** P3 11111111**								B7
A8	*** IE 00000000**	**SADDR 00000000**							AF
A0	* P2 11111111								A7
98	* SCON 00000000	* SBUF XXXXXXXX							9F
90	* P1 11111111								97
88	* TCON 00000000	* TMOD 00000000	* TL0 00000000	* TL1 00000000	* TH0 00000000	* TH1 00000000			8F
80	* P0 11111111	* SP 00000111	* DPL 00000000	* DPH 00000000				***PCON ** 00XX0000**	87

* = Found in the 8051 core (See 8051 Hardware Description in the Embedded Controller Handbook for explanations of these SFRs).

** = See description of PCON SFR. PCON also contains two additional bits, GF1 and GF0, which are general purpose flag bits.

September 1988

Small DC Motor Control

JAFAR MODARES
ECO APPLICATIONS